

# LiteMat: a scalable, cost-efficient inference encoding scheme for large RDF graphs

Olivier Curé, Hubert Naacke, Tendry Randriamalala, Bernd Amann  
LIP6 CNRS UMR 7606

Sorbonne Universités, UPMC Univ Paris 06, F-75005, Paris, France

Email: {firstname.lastname}@lip6.fr

**Abstract**—The number of linked data sources and the size of the linked open data graph keep growing every day. As a consequence, semantic RDF services are more and more confronted with various "big data" problems. Query processing in the presence of inferences is one them. For instance, to complete the answer set of SPARQL queries, RDF database systems evaluate semantic RDFS relationships (subPropertyOf, subClassOf) through time-consuming query rewriting algorithms or space-consuming data materialization solutions. To reduce the memory footprint and ease the exchange of large datasets, these systems generally apply a dictionary approach for compressing triple data sizes by replacing resource identifiers (IRIs), blank nodes and literals with integer values. In this article, we present a structured resource identification scheme using a clever encoding of concepts and property hierarchies for efficiently evaluating the main common RDFS entailment rules while minimizing triple materialization and query rewriting. We will show how this encoding can be computed by a scalable parallel algorithm and directly be implemented over the Apache Spark framework. The efficiency of our encoding scheme is emphasized by an evaluation conducted over both synthetic and real world datasets.

## I. INTRODUCTION

The Resource Description Framework (RDF) data model is the W3C standard for representing metadata in the Web of Data and the Semantic Web. This format is used to form very large graphs, such as those found in Linked Data sources and Linked Open Data (LOD), which range in hundreds of millions to billions of RDF triples. With such workloads, RDF database systems are now facing "big data" problems. In this work, we focus on issues related to SPARQL query processing over large RDF data with rich semantics. One aspect that differentiates RDF data, and Knowledge bases (KB) in general, from standard relational and NoSQL systems, e.g., graph databases such as Neo4J and Titan, is the ability to reason over the represented information using associated knowledge. Such knowledge, contained in ontologies, is generally expressed using RDF Schema (RDFS) or Web Ontology Language (OWL) W3C recommendations. From a query processing point of view, to achieve completeness of result sets, RDF query processors have to integrate information that is inferred using these ontologies. We can distinguish two main approaches to support this kind of inference. The first approach consists of materializing all derivable triples in the RDF store before evaluating queries. The second approach consists of rewriting each submitted query into an extended

query including semantic relationships from the ontologies. Both methods have advantages and drawbacks.

On one hand, materialization implies a possibly long loading time due to running reasoning services during a data pre-processing phase. This generally drastically increases the size of the stored data and imposes specific dynamic inference strategies when data is updated. The advantage is that it ensures good query performance.

On the other hand, query rewriting avoids costly data pre-processing, storage extension and complex update strategies but induces slow query response times since all the reasoning tasks are part of a complex query pre-processing step.

In the worst case the computational complexity associated to data materialization and query rewriting processing are exponential in the size of the original data and query respectively. Some existing techniques [11] propose polynomial solutions in specific situations. We now provide an example to make these two approaches more concrete.

*Example 1:* We consider a simple example based on the following extract of the LUBM ontology [8]. The concept hierarchy (TBox, short for Terminological Box) is limited to the following axioms:

(1) *Professor*  $\sqsubseteq$  *FacultyMember*

(2)  $\exists teaches.\top \sqsubseteq$  *FacultyMember*

where  $\sqsubseteq$  denotes the subsumption relationship, i.e., a *Professor* is a subconcept of *FacultyMember*. The domain of the property *teaches* is the concept *FacultyMember*. The ABox (short for Assertional Box), is a set of facts and consists of the following RDF triples:

(3) *bernd type Professor*. (4) *hubert teaches course1*.

The SPARQL query aiming to retrieve all *FacultyMember* instances is expressed as follows:

*SELECT ?x WHERE { ?x type FacultyMember }*

The complete and correct answer set with respect to the TBox contains both *bernd* and *hubert* but without any inference, the query would return an empty result set since none of the instances, i.e., *bernd*, *hubert* or *course1*, are explicitly typed with the concept *FacultyMember*. In the materialization approach the ABox would be extended to contain the following triples:

(5) *bernd type FacultyMember*.

(6) *hubert type FacultyMember*.

where (5) is deduced from (1) and (3) while (6) is derived from (2) and (4).

A complete and sound answer set can be retrieved with the following reformulated query:

```
SELECT ?x WHERE {{?x type FacultyMember.}
UNION {?x teaches ?y.} UNION {?x type Professor.}}
```

In this article, we present the LiteMat data encoding and SPARQL query evaluation approach which aims at finding an efficient trade-off between materialization and query rewriting that provides complete answer sets considering the RDFS entailment regime. The LiteMat approach builds on a semantic-aware RDF data encoding which allows to reduce the amount of materialized information with minimal query reformulation and processing cost. The approach is based on a clever encoding of the ontology elements of the KB and only requires some functions to check whether a variable binding belongs to a numerical interval. We will demonstrate that LiteMat can reduce the number of triples of an original dataset.

Most RDF stores, e.g. [10], adopt an encoding approach to compress RDF triples. This is motivated by the fact that components of RDF triples, i.e., subjects, predicates and objects, mainly correspond to Internationalized Resource Identifiers (IRI) or literals. They correspond to long strings of characters and the number of their occurrence can be quite important in real world cases. Thus replacing them with integer values permits to obtain a much more compact representation of a set of triples. Blank nodes are encoded using a similar approach.

Most RDF systems use dictionary-based encoding for long IRI resource identifiers but do not consider semantic aspects when attributing integer values to RDF entities (see [5] for more details on this topic). That is, they do not consider the ontology at this processing stage. As a result, no distinctions are made between an instance or concept/predicate IRI. Our solution uses that distinction and is based on a clever encoding of the TBox which is implied in the encoding of the ABox.

Given the size of currently available RDF datasets, this encoding step can result in a performance bottleneck. To speed up this processing, a parallel computation may be necessary. Once the encoding is performed, two operations are needed to handle an encoded dataset: *locate* and *extract*. The *locate* operation takes as input a string (IRI, blank node identifier or literal) and provides the corresponding id. The *extract* operation is provided with an id and returns a string value. These operations generally handle key/value pairs.

The main contributions of this paper are to propose an encoding algorithm for the TBox, to present a generic scalable algorithm for the encoding of ABoxes, to provide a lite materialization strategy together with its query processor supporting SPARQL's RDFS entailment regime. Finally, to tackle very large graphs, we evaluate our implementation over the Apache Spark framework using synthetic and real world use cases.

## II. BACKGROUND KNOWLEDGE

### A. RDF, SPARQL, RDFS entailment regime

RDF is a schema-free data model that supports the description of data on the Web. It is usually considered as the cornerstone of the Semantic Web and the Web of Data. Assuming disjoint infinite sets  $U$  (RDF IRI references),  $B$  (blank nodes) and  $L$  (literals), a triple  $(s, p, o) \in (U \cup B) \times U \times$

$(U \cup B \cup L)$  is called an RDF triple with  $s$ ,  $p$  and  $o$  respectively being the subject, predicate and object. We now also assume that  $V$  is an infinite set of variables and that it is disjoint with  $U$ ,  $B$  and  $L$ . We can recursively define a SPARQL<sup>1</sup> triple pattern as follows: (i) a triple  $tp \in (U \cup V) \times (U \cup V) \times (U \cup V \cup L)$  is a SPARQL triple pattern, (ii) if  $tp_1$  and  $tp_2$  are triple patterns, then  $(tp_1, tp_2)$  represents a group of triple patterns that must all match,  $(tp_1 \text{ OPTIONAL } tp_2)$  where  $tp_2$  is a set of patterns that may extend the solution induced by  $tp_1$ , and  $(tp_1 \text{ UNION } tp_2)$ , denoting pattern alternatives, are triple patterns and (iii) if  $tp$  is a triple pattern and  $C$  is a built-in condition then the expression  $(tp \text{ FILTER } C)$  is a triple pattern that enables to restrict the solutions of a triple pattern match according to the expression  $C$ . The SPARQL syntax follows the select-from-where approach of SQL queries. The SELECT clause specifies the variables appearing in the result set of the query.

The SPARQL 1.1 W3C recommendation specifies various entailment regimes which define the evaluation of SPARQL triple patterns by means of subgraph matching. In this work we are interested in the RDFS entailment regime. It is based on the set of reasoning rules of the RDFS language<sup>2</sup>. We concentrate on property and class subsumption, domain and range of properties. They have been selected due to their support of the most frequent RDFS inferences.

### B. Apache Spark

Apache Spark [15] is a cluster computing framework based on a shared nothing architecture. Just like Apache Hadoop, Spark enables parallel computations on unreliable machines and automatically handles locality-aware scheduling, fault tolerance and load balancing. While both systems are based on a data flow computation model, Spark is more efficient than Hadoop for applications requiring to reuse working datasets across multiple parallel operations. This efficiency is due to Spark's Resilient Distributed Dataset (RDD) [14], a distributed, lineage supported fault tolerant memory abstraction that enables one to perform in-memory computations (when Hadoop is mainly disk-based). The Spark API also eases the programming task by integrating functions which are not natively supported in Hadoop, e.g., join, filter. The design and implementation of Spark started at UC Berkeley's AMPLab and at the time of writing this paper, it is considered to be the Apache project with the most committers.

## III. KNOWLEDGE BASE ENCODING

The encoding principle of our approach is similar to the ones encountered in most RDF Stores. A first phase creates a dictionary corresponding to a bijective function mapping long terms (IRI, blank node identifier, literal) to short identifiers (integer). The dictionary is then managed using the *locate* and *extract* operations previously introduced.

The originality of our approach is that we distinguish the TBox encoding from the ABox one. For the former, we

<sup>1</sup><http://www.w3.org/TR/rdf-sparql-query/>

<sup>2</sup><http://www.w3.org/TR/rdf11-nt/>

propose a semantic-aware encoding which supports the lite materialization approach. While this phase is performed on a single machine of the cluster, the ABox encoding is computed in parallel using Spark. The values we assign for entities of the TBox and ABox can overlap, i.e., the same value can be given to an individual, a property and a concept. This does not cause any problems in our system due to context awareness at query processing time. For instance, we know that each identifier at the second position of an RDF triple is necessarily a property and that objects associated to an `rdf:type` property are necessarily concepts. All other identifiers in the ABox correspond to individuals.

#### A. TBox encoding

1) *Principle*: Our TBox encoding principle applies to both the concept and property hierarchies, which we denote as entities in the following. The idea is to assign an integer value to each entity such that for a sub-entity B of an entity A ( $B \sqsubseteq A$ ), B's assigned identifier ( $idB$ ) is comprised in an interval that can be easily and automatically compute from A's identifier ( $idA$ ). That is  $idB \in ]idA, idA + \epsilon[$  where  $\epsilon$  depends on the number of direct sub-entities of the entity A. This encoding scheme applies recursively to the complete entity hierarchy.

This encoding scheme covers both the `rdfs:subClassOf` and `rdfs:subPropertyOf` associated entailments. To support inference rules related to `rdfs:domain` and `rdfs:range`, we define two additional data structures, one for each RDFS property, consisting of a key/value pair. The key corresponds to the property identifier and the value contains the set of this property's domain (resp. range) concept identifiers.

2) *Implementation*: In order to grasp accurate entity hierarchies, we use an OWL reasoner to infer concept classifications. This implies that we are representing hierarchies that are not limited to the interpretation of the RDFS language. Nevertheless, based on this representation, we only consider the RDFS entailment regime, e.g., currently our system is not able to handle inference rules related to transitive properties.

HermiT [9], the OWL reasoner we are using for the encoding, does not support a distributed computing approach. We thus generate the encoding of a TBox on a single machine. The algorithm represents entity identifiers as vectors of bits and consists of two phases. In the first phase, a top-down navigation of the inferred entity hierarchy, e.g., for concepts, is adopted. Intuitively, we start from the `owl:Thing` concept down to all subconcepts. Given an entity A, we first compute the number N of direct sub-entities. These sub-entities will be encoded over  $\lceil \log_2(N+1) \rceil$  bits. This is performed recursively until all entities in the TBox are assigned to an identifier. It is guaranteed at the end of this first phase that, for 2 entities A and B with  $B \sqsubseteq A$ , the prefix of  $idB$  matches with the encoding  $idA$ .

At the end of this first step, all entities have been given a prefix value. Table I presents an extract of the encoding of the LUBM ontology. For instance, in LUBM, the `Schedule` and `AssociateProfessor` are respectively given identifiers

Encoding	Concept label
0 000 0000000000	Thing
0 001 0000000000	Schedule
0 010 0000000000	Organization
0 011 0000000000	Publication
0 100 0000000000	Person
0 100 0100000000	TeachingAssistant
0 100 1000000000	Student
...	
0 100 1110010011	AssociateProfessor
...	
0 101 0000000000	Work

TABLE I. EXTRACT OF THE LUBM CONCEPT ENCODING

'0001' and '01001110010011'. Given that `owl:Thing` is always assigned the value 0, we can see that its 5 direct subconcepts (`Schedule`, `Organization`, `Publication`, `Person` and `Work`) are encoded over 3 bits. Moreover, `AssociateProfessor` shares the prefix '0100' with its indirect `Person` ancestor. In order to support our interval operations over identifiers, we need to represent these bit vectors over the same number of bits. The total number of bits corresponds to the encoding of the longest entity identifier, i.e., 14 bits for the `AssociateProfessor` or any of its siblings. Note that encodings for DBPedia and Wikidata's concept hierarchies required respectively 27 and 102 bits, justifying the use of vector bits for the latter. This second step handles the identifier completion by appending '0' on rightmost bits.

#### B. ABox encoding

1) *Principle*: The encoding of the ABox is more straightforward than the TBox encoding since it does not require an external software component, i.e., a reasoner. The idea is to provide a unique integer identifier to all ABox entries that have not been assigned an identifier during the TBox encoding. Since we are not giving any 'meaning' to these values, the encoding can be performed in parallel. This presents a particular advantage due to the size of the ABox which can be orders of magnitude larger than the TBox.

2) *Implementation*: For the ABOX encoding we will benefit from the Spark framework by storing most of the processed datasets, i.e., the portion corresponding to the ABox, in main-memory and by taking advantage of a rich API of operators including join, union, duplicate elimination. In a first stage, the dataset is uploaded from the Hadoop Distributed File System (HDFS) and partitioned across the cluster. The system identifies the subject/object and possible property terms which are not covered by the TBox encoding. Then each distinct entry is assigned an identifier in a distributed manner. Intuitively, we create an array that contains the number of subjects and objects contained in each partition. These values are summed up so that each partition can be associated with a disjoint identifier interval. Then each partition provides identifiers to all its members ranging over the partition's identifier interval. This set of subject/object (resp. properties) is then unioned with the concept (resp. property) encoding emanating from the TBox.

Given these two maps, the original string-based dataset is encoded via some join operations whose execution can be optimized by replicating the maps over the cluster nodes, i.e.,

storing the maps in the main-memory of all executors of the cluster and thus preventing network communications. In our evaluation, the property map is broadcasted while for some cases, e.g., Wikidata (Table II), broadcasting the concept map would be counterproductive due to its large memory footprint.

#### IV. LITE MATERIALIZATION

Our TBox encoding scheme provides the nice property of retaining only the most specific concept in the hierarchy while still being able to perform some subsumption inferences. In this context, our LiteMat approach aims to minimize the amount of types assigned to each instance in the dataset. This may be useful since some real world datasets are known to have several types for a given instance, e.g., an average of 8 different types are provided on DBPedia individuals. The information stored in our TBox encoding data structures enables to perform this materialization.

The parallel algorithm proceeds in 2 steps. In the first step, for each individual  $I$ , the system gathers all explicit and implicit types where the former correspond to types already stored in the original datasets and the latter correspond to the ones that can be derived from the domain and range of RDFS properties involved in  $I$ 's assertions. This set of types is henceforth denoted as candidate concepts. For instance, in Example 1, the domain of `teaches` enables to attribute the `FacultyMember` type to the *hubert* instance.

In Figure 1, we represent a set of such concepts in the context of their subsumption hierarchies. Each node represents a TBox concept with the black ones being the candidate concepts and the white ones being their super concepts. The plain lines represent subsumption hierarchies with the lower node always being the subconcept. To minimize the number of types for a given instance, the system needs to retain only the Most Specific Concepts (MSC) in each subsumption branch of candidate concepts. This corresponds to identifying the nodes going through the dashed line in Figure 1. Note that during this identification process, we are not considering the complete concept hierarchy but only the candidate concepts and the transitive closure of their super concepts.

Our system performs this operation in one pass using our TBox encoding data structures. Intuitively, set of candidate concepts of an instance  $I$  is ordered in descending order on the values of their identifiers. This is motivated by our encoding scheme which ensures that a subconcept has a greater value than its super concept. The first concept in that list is stored in the MSC set and subsequent candidates are checked for insertion in the MSC set. A candidate is inserted in the MSC set if none of the concepts in the MSC set belongs to its subsumption interval.

A concept's 'subsumption interval' is computed using the concept id and 3 meta data corresponding to its total encoding length (`codeLength`), the position at which this concept encoding starts in the bit vector (`start`) and the length of its encoding (`localLength`). Given these information, the following *bound* function returns the upper bound of a concept.

```
def bound(id : String, start: Int,
          localLength: Int, codeLength: Int)
```

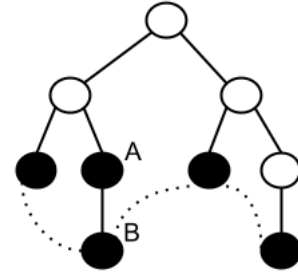


Fig. 1. Candidate concepts in their hierarchy

```
: (String) = {
  val shift = codeLength -
              (start+localLength)
  val prefix = id >> shift
  val upperBound = (prefix+1) << shift
  return upperBound
}
```

In Figure 1, the concept A is not retained in the MSC set since B is more specific. Consider the following setting: A and B have respective identifiers: 20 (00010100) and 22 (00010110), the encoding length is 8 bits, the encoding of A starts at bit 3 and is encoded using 3 bits. Then the subsumption interval of A is  $[20, 24[$ . This justifies that A is not in the final MSC since B with an identifier of 22 belongs to that interval.

We finally would like to highlight that the lite materialization may also have the positive effect of diminishing the original dataset by removing some unnecessary concepts (one of our experimentation highlights such a situation on the Wikidata real world dataset).

#### V. QUERY PROCESSING

This section details our evaluation method of conjunctive SPARQL queries using the encoding scheme described in Section III. This method guarantees the completeness of the query result. It considers the RDFS class and property hierarchies, i.e., considers all triples that match a graph pattern according to these hierarchies. More precisely, the triple matching procedure takes into account the following semantic inference rules:

- a triple  $(s, p, o)$  matches the triple pattern  $(?x, prop, ?y)$  if  $p$  is a sub-property of  $prop$  (by definition  $p$  is a subproperty of itself).
- a triple  $(a, rdf:type, b)$  matches the triple pattern  $(x, rdf:type, c)$  if  $b$  is a subtype of  $c$  (by definition,  $b$  is a subconcept of itself).

These rules can be efficiently evaluated by the semantic-aware encoding described in Sec. III. It allows the query processor to compare straightforwardly the values of  $p$  and  $prop$  (and respectively the values of  $b$  and  $c$ ), based on the encoded values and the previously defined *bound* function

(Sec. IV):

$$p \text{ is a subproperty of } prop \Leftrightarrow prop \leq p < bound(prop) \quad (1)$$

$$b \text{ is a subtype of } c \Leftrightarrow c \leq b < bound(c) \quad (2)$$

Note that we implement subproperty and subtype matching as a single comparison operation without testing every existing subproperty and subtype. This obviously saves computation cost.

Our query plan generation method targets the Apache Spark distributed and parallel computing platform for processing queries. As explained in Sec. II-B, Spark provides efficient methods to manipulate distributed datasets in parallel. We use the following Spark operations applied to datasets:

- `dataset.filter(predicate)`: select the subset of the dataset that satisfies a predicate.
- `dataset.map(function)`: apply a function on each element of the dataset.
- `dataset.join(dataset)`: join two datasets using an equality predicate. It requires that the structure of the elements in each dataset is a pair. For instance (a,b) join (c,d) produces a result if a=c.

It has been showed in [13] that these operations are sufficient to express any conjunctive query. With this in mind, we describe the translation of conjunctive SPARQL query into an algebraic expression.

**Locate step.** In a typical use case, a SPARQL engine calls the *locate* function to transform the constants of a triple pattern into numerical values. In our case, we translate every term (property, type and literal) of the query into its corresponding encoded value.

*Example 2:* The execution of the *locate* function over Query #4 in A would yield the following triple patterns:

```
?x 0 1044643840. ?y 0 335544320.
?x 1145044992 ?y.
```

**Matching step.** We translate each query triple, into a filter operation that implements the matching. For clarity, we detail below the resulting algebraic expression written in (simplified) Scala programming language. The name *triples* denotes the dataset.

(?x, prop, ?y) is translated into

```
triples.filter( (s,p,o) =>
  prop <= p && p < bound(prop) )
```

(?x, rdf:type, c) is translated into

```
triples.filter( (s,p,o) =>
  p == type && c <= o && o < bound(c) )
```

**Conjunction step.** We translate the conjunction of two triple patterns into a join expression. We denote t1 (resp. t2) the expression resulting from the translation of the first (resp. the second) triple pattern

(?x, prop<sub>1</sub>, ?y) . (?y, prop<sub>2</sub>, ?z) is translated into

```
t1.map( (s,p,o) => (o, (s,p)) ).
  join(t2.map( (s,p,o) => (s, (p,o)) )
```

The map function allows to specify on which variable the equality join must apply. Our translation method supports

all the join cases: subject-subject, subject-object, and object-object. We handle several conjunctions by successively applying that translation on each pattern. Note that the finding an optimal join ordering for triple patterns is beyond the scope of this article.

**Extract step.** The last query processing step consists of calling the *extract* function to translate the result set (so far expressed with numerical values) into an end-user understandable form, i.e., IRIs, literals and possibly blank node identifiers.

## VI. EVALUATION

### A. Computational environment

The evaluation was conducted on a cluster consisting of 15 DELL PowerEdge R410 running a Debian GNU/Linux distribution with a 3.16.0-4-amd64 kernel version. Each machine has 64GB of DDR3 RAM, two Intel Xeon E5645 processors. Each processor is constituted of 6 cores running at 2.40GHz and allowing to run two threads in parallel (hyper threading). Hence, the number of virtual cores amounts to 24. Concerning storage, each machine is equipped with a 900GB 7200rpm SATA disk. The machines are connected via a 1GB/s Ethernet network adapter. We used Spark version 1.4.1 and implemented all experiments in Scala, using version 2.10.4. The Spark configuration of our evaluation enables to run our prototype on a subset of the cluster corresponding to 300 cores and 50GB of RAM per machine.

### B. Datasets and queries

We have selected 2 synthetic and 2 real world knowledge bases. The synthetic datasets correspond to instances of the LeHigh University Benchmark, a well-established Semantic Web set of tools composed of an ontology, a set of queries and a data generator. The knowledge bases have respectively been configured with 1000 and 10000 universities, resp. denoted LUBM1K and LUBM10K. The real world datasets correspond to open source DBPedia and Wikidata RDF dumps.

In table II, we present the main characteristics of the associated ontologies. Concerning DBPedia and Wikidata, the ontologies respectively correspond to DBPedia\_2014<sup>3</sup> and the union of wikidata-taxonomy and wikidata-properties<sup>4</sup>. We can observe that they differ in terms of their size (from 15KB to 78MB), number of concepts (from 43 to over 210K), properties (from around 30 to over 3K) and number of axioms related to property domain and range (from none to around 2K). The main characteristics of the ABoxes are reported in the first two columns of Table III.

For the query processing evaluation, we choose a subset of the LUBM benchmark queries that allow us to highlight the benefits of our LiteMat approach.

### C. Results

In this section, we compare the encoding and query performance of our solution with a baseline solution based on a full materialization using a standard TBOX-unaware dictionary encoding.

<sup>3</sup><http://oldwiki.dbpedia.org/Downloads2014>

<sup>4</sup><http://tools.wmflabs.org/wikidata-exports/rdf/exports/20140526/>

Ontology	Size	#Concepts	#Prop. (Obj.+Datatype)	#Domain axioms	#Range axioms	Total time in sec.
LUBM	15KB	43	32 (25+7)	21	18	0.7
DBPedia	2.3MB	814	3,035 (1,310+1,725)	1,076	997	3.7
Wikidata	78MB	213,958	353 (255+98)	0	0	122

TABLE II. MAIN FEATURES AND ENCODING TIME OF THE EVALUATED ONTOLOGIES

Dataset	#Triples *10 <sup>6</sup>	SAE time (sec.)	SAE throughput (triples/sec.)	OBE time (sec.)	OBE throughput (triples/sec.)
LUBM1K	133.5	280.5	476,199	113.6	1,157,486
LUBM10K	1,334.7	3,746.5	356,248	1,755.8	759,378
DBPedia	79.1	282.2	280,943	128	617,050
Wikidata	242.1	1,334.8	181,394	477.2	507,386

TABLE III. COMPARISON OF THE STANDARD ABOX ENCODING (SAE) WITH THE ONTOLOGY-BASED ENCODING (OBE)

1) *Encoding efficiency*: In order to evaluate LiteMat’s encoding strategy, we present the performances of both our TBox and ABox encoding. To evaluate the ABox encoding, we compare the standard ABox-only encoding (denoted SAE) which ignores any TBox knowledge, versus our ontology-based encoding solution (denoted OBE). SAE is completely computed in parallel over Spark. OBE is composed of an ontology encoding (implemented in Java and using the HermiT reasoner and the OWLAPI) and an ABox encoding (which runs over Spark in a manner quite similar to SAE) steps.

The rightmost column of Table II presents the duration of the ontology encoding, i.e., processing the concept and property hierarchies as well as the structures for the domain and range RDFS properties, for the three ontologies under test. We report relatively fast execution times for all ontologies (up to 122 seconds for the largest ontology containing more than 200K concepts and over 300 properties).

We also report the encoding duration and throughput (i.e., the number of RDF triple statements encoded per second) of the two encoding solutions in Table III. Note that the reported times for OBE includes the TBox encoding time (Table II). First we observe that both OBE and SAE solutions yield high throughput compared to the state-of-the-art solution reported in [12], thanks to our highly parallel implementation of the two solutions over the Spark platform.

Second, we observe that the OBE is up to 2.8 times faster than the SAE. In fact, for SAE, all individuals, concepts and properties have to be encoded. While for OBE, concepts and properties have already been encoded and are just uploaded for ABox encoding. The duration difference between the two approaches is explained by the high number of *rdf:type* triples in most RDF datasets.

Moreover, for OBE, the size of the ontology unsurprisingly impacts the duration: although LUBM1K is 1.6 times larger than the DBPedia dataset, its encoding time is 12% shorter.

Finally, we observe rather promising scalability of our OBE solution for growing datasets: the throughput is only decreasing by 35% when the LUBM dataset size is ten times larger (ranging from 133M to 1.3B triples).

2) *Evaluation of materialization strategies*: In this section, we compare our LiteMat materialization approach with a standard full materialization solution. The results are respectively reported in Tables IV and V. First, for small dataset the duration is about the same, we observe a difference for larger

KBs where the full materialization is much longer (between 1.5 and 2 times longer). Second, for LUBM, the dataset size remains almost the same. Indeed, the number of extra triples added during domain/range inference is slightly the same as the number of triples removed when retaining only the most-specific concepts. Third, Considering the full materialization, unsurprisingly, the relative increase of the dataset size is ranging from 13% for DBPedia and up to almost 58% for Wikidata. This increase in size is directly related to the depth of ontology entity hierarchies, e.g., the DBPedia and Wikidata have several concept branches of depth larger than 6.

3) *Query processing benefits*: We conduct experiments to quantify the benefits of LiteMat on querying large RDF databases. We implement all the queries (detailed in Appendix A) in Scala using the translation method proposed in Sec. V. We execute queries  $Q_1$  to  $Q_4$  on the LiteMat and on the full materialization of the LUBM10K dataset. For baseline comparison, we also execute the rewritten queries  $Q'_1$  to  $Q'_4$  on the original LUBM10K dataset. We first check for query completeness: each query consistently returns the same result set on the lite, full, and original dataset. Then, we report on Table VI the response times of  $Q_i$  queries for LiteMat and or full materialization, and of  $Q'_i$  queries for the no materialization database.

We observe that our solution (LiteMat) outperforms the baseline approach (no mat.) from 11% (for  $Q_3$ ) up to 28% (for  $Q_1$ ). This benefit is rather impressive considering that we implement the baseline approach in an optimized way using a conjunction of “OR” subqueries instead of a costly union of conjunctive subqueries.

On the opposite, we observe that query  $Q_4$  containing 3 triple patterns, performs slower than its rewriting, because in this particular case, the rewriting consists of the union of an empty subquery (indeed, the original dataset does not contain any triple with the *Chair* concept) and another simple subquery having only 2 triple patterns. However, this is not a limitation of our approach since one can always simplify the query before executing it on the LiteMat database.

As expected, full materialization yields relatively poor performances because of the overhead implied by accessing a larger database. The results show that LiteMat allows for fast query processing (thanks to the encoding) and the ability to store most of processed data in main-memory (a property due the Spark framework).

Dataset	Duration (sec.)	Added triples (%)	Deleted triples (%)	throughput (triples/sec.)
LUBMIK	172.7	0	0	466,552
LUBMI0K	1,755.8	0	0	307,139
DBPedia	29.2	0	0	478,129
Wikidata	859.2	0	0.04	77,240

TABLE IV. ONTOLOGY-BASED ENCODING WITH LITE MATERIALIZATION

Dataset	Duration (sec.)	Added triples (%)	throughput (triples/sec.)
LUBMIK	176.6	38.15	460,917
LUBMI0K	3,136.8	38.2	272,796
DBPedia	57	13.6	356,348
Wikidata	1,482	57.9	123,571

TABLE V. ONTOLOGY-BASED ENCODING WITH FULL MATERIALIZATION

Query	Lite mat.	Full mat.	No mat.
Q1	15.6	20.2	21.8
Q2	15.6	17.2	21.6
Q3	72.2	99.2	81.4
Q4	50.2	79.8	27.2

TABLE VI. QUERY TIMES (IN SEC.) FOR LITE/FULL/NO MATERIALIZATION

## VII. RELATED WORKS

The first set of related works concerns semantic-aware RDF data encoding. Most existing works in this field have been influenced by [1] which proposes to encode subsumption hierarchies with numeral values. This method proposes to encode the nodes of a tree in such a way that all the numerical values associated to subnodes of a supernode all range within an interval that can be computed from the supernode value. Although the work of [1] is essentially focusing on the hierarchies themselves, the approach was extended in [11], with the so-called semantic index technique, to Ontology-Based Query Answering (OBQA) for the RDFS fragment of DL-Lite [3]. This later work is not providing any algorithmic details and does not consider the parallel computation of the dictionaries and of the dataset transformation. The work in WaterFowl [6] also influenced the LiteMat project. Both systems are adopting a binary encoding of the elements of the TBox. In WaterFowl, this was mainly motivated by the use of Wavelet tree data structures whose nodes are composed of bit vectors, to store encoded RDF datasets. Rather than focusing on identifier intervals at query answering time, WaterFowl uses a binary prefix encoding scheme to navigate in wavelet trees. Like the two other systems in this section, WaterFowl does not scale for large datasets due to the lack of a parallel computing approach.

Another set of related works concerns the distributed encoding of RDF datasets. The most recognized systems in this field are either using a dedicated computing environment [7], the Map Reduce model [12] or a parallel language running over a shared nothing architecture [4]. The approach presented in [7] uses a shared memory Cray XMT multi-million dollars machine to perform parallel hashing based on the linear probing scheme. The evaluation emphasizes that the method is efficient due to main-memory storage and performances are linear with the number of used cores. Nevertheless, the algorithm is specific to the shared memory architecture of the machine. The work of [12] computes RDF encodings on top of

the Hadoop framework using a standard cluster of commodity hardware (which was it the case of the aforementioned system). The approach uses a sampling approach to encode the most popular terms which are cached in main memory. The encoding is performed through a set of map and reduce functions which can imply costly network communications. Moreover, the use of Hadoop's MapReduce implies I/O contention due to the intensive use of disk accesses. Finally, [4] proposes an implementation using the X10 parallel programming language consisting of 3 steps: fragmentation of the data in equal-size chunks (based on terms hash value), encoding of these entries of these chunks are computed locally on each node of the cluster, much like our ABox encoding. Note that none of these distributed encoding approaches make a distinction between the TBox and the ABox. Hence, their semantic-unaware encoding is not able to support inference optimization. Moreover, our Spark-based implementation has the advantages of being easier to develop for and maintain due to its user-friendly, rich set of APIs.

## VIII. CONCLUSION

In this paper, we have presented LiteMat, a data encoding scheme and query evaluation approach for the SPARQL query language. The main contribution of the system is an efficient and complete RDF query answering solution respecting the SPARQL's RDFS entailment regime. By taking the most of our clever TBox encoding, we are able to reduce the amount of materialized data and to limit the effort of query reformulation. Conducted evaluations emphasized the efficiency of the approach in terms of data storage size as well as encoding and query processing performances. Moreover, except for the TBox encoding, all processing steps of our system are implemented over the parallel computing framework Apache Spark, and thus produce high triple statements per second processing rates.

We consider that there is plenty of room for improvement and optimization for our system. For instance, some future works will concentrate on query optimization using some

heuristics and statistics which can be computed during the ABox encoding time. We are also planning to support a more expressive entailment regime at query run-time, e.g., RDFS+ [2] which provides to specify properties characteristics (inverse, symmetric and transitive) and equality between individuals, classes and properties.

#### ACKNOWLEDGMENT

This work has been partially supported by the ARESOS project from CNRS Program MASTODONS.

#### REFERENCES

- [1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989.*, pages 253–262, 1989.
- [2] D. Allemang and J. A. Hendler. *Semantic Web for the Working Ontologist - Effective Modeling in RDFS and OWL, Second Edition.* Morgan Kaufmann, 2011.
- [3] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. Autom. Reasoning*, 39(3):385–429, 2007.
- [4] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. Efficiently handling skew in outer joins on distributed systems. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Chicago, IL, USA, May 26-29, 2014*, pages 295–304, 2014.
- [5] O. Curé and G. Blin. *RDF Database Systems: Triples Storage and SPARQL Query Processing, 1st Edition.* Morgan Kaufmann, Nov. 2014.
- [6] O. Curé, G. Blin, D. Revuz, and D. C. Faye. Waterfowl: A compact, self-indexed and inference-enabled immutable RDF store. In *The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings*, pages 302–316, 2014.
- [7] E. L. Goodman, E. Jimenez, D. Mizell, S. Al-Saffar, B. Adolf, and D. J. Haglin. High-performance computing applied to semantic databases. In *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II*, pages 31–45, 2011.
- [8] Y. Guo, Z. Pan, and J. Heflin. Lbun: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
- [9] B. Motik, R. Shearer, and I. Horrocks. Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.
- [10] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.
- [11] M. Rodriguez-Muro and D. Calvanese. High performance query answering over dl-lite ontologies. In *KR*, 2012.
- [12] J. Urbani, J. Maassen, N. Drost, F. J. Seinstra, and H. E. Bal. Scalable RDF data compression with mapreduce. *Concurrency and Computation: Practice and Experience*, 25(1):24–39, 2013.
- [13] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 13–24, 2013.
- [14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.
- [15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.

#### APPENDIX

Following are the SPARQL queries executed over both the LUBM1K and LUBM10K of our experimentation.

##### A. Query $Q_1$ : Inferences over the concept hierarchy

```
SELECT ?x WHERE { ?x rdf:type Professor. }
```

and its rewriting  $Q'_1$ :

```
SELECT ?x WHERE { { ?x rdf:type Professor. }
  UNION { ?x rdf:type AssistantProfessor. }
  UNION { ?x rdf:type AssociateProfessor. }
  UNION { ?x rdf:type Chair. }
  UNION { ?x rdf:type Dean. }
  UNION { ?x rdf:type Faculty. }
  UNION { ?x rdf:type VisitingProfessor. }
  UNION { ?x rdf:type FullProfessor. } }
```

##### B. Query $Q_2$ : Inferences over the property hierarchy

```
SELECT ?x ?y WHERE { ?x memberOf ?y. }
```

and its rewriting  $Q'_2$ :

```
SELECT ?x ?y WHERE { { ?x memberOf ?y } UNION
  { ?x worksFor ?y } UNION { ?x headOf ?y } }
```

##### C. Query $Q_3$ : Inferences over the concept and property hierarchies

```
SELECT ?x ?y WHERE { ?x rdf:type Professor.
  ?x memberOf ?y. }
```

The rewriting, denoted  $Q'_3$ , of this query corresponds to the union of the cartesian product of the WHERE clauses of queries  $Q_1$  and  $Q_2$ . The query thus contains the union of 18 filters. For space limitation, we only display a subset of them:

```
SELECT ?x WHERE { { ?x rdf:type Professor.
  ?x memberOf ?y. } UNION { ?x rdf:type
  Professor. ?x worksFor ?y. } UNION { ?x
  rdf:type Professor. ?x headOf ?y. } UNION
  { ?x rdf:type AssistantProfessor.
  ?x memberOf ?y. } ... }
```

##### D. Query $Q_4$ : Inferences over the property hierarchy

This query is a generalized pattern of the query  $Q_{12}$  from the LUBM benchmark.

```
SELECT ?x WHERE { ?x rdf:type Chair.
  ?y rdf:type Department. ?x worksFor ?y. }
```

and its rewriting  $Q'_4$ :

```
SELECT ?x WHERE { { ?x rdf:type Chair.
  ?y rdf:type Department. ?x worksFor ?y. }
  UNION { ?x headOf ?y.
  ?y rdf:type Department. } }
```